Chapitre 5 : Arbres Binaires de Recherche

Les arbres binaires de recherche sont des arbres adaptés à la représentation chaînée. Grâce à leur structure, ils sont très efficaces lorsqu'on veut effectuer avec les mêmes performances la suppression, l'ajout et la recherche d'éléments.

Méthode de recherche arborescente :

- comparer l'élément avec un nœud de l'arbre permet d'orienter la recherche dans l'arbre.
- on parcourt rarement tout l'arbre (plus économique).
- gain significatif de complexité (souvent logarithmique en la taille de l'arbre).

1. Définition des arbres binaires de recherche (ABR)

On représente une collection ordonnée de N éléments par un arbre binaire étiqueté de N nœuds. La répartition des éléments dans l'arbre permet de guider la recherche en faisant des comparaisons. La structure et les propriétés de l'ABR doivent être conservées à l'ajout et à la suppression d'éléments pour lui conserver ses propriétés.

Un arbre binaire de recherche un arbre tel que, ***pour tout nœud courant de l'arbre*** :

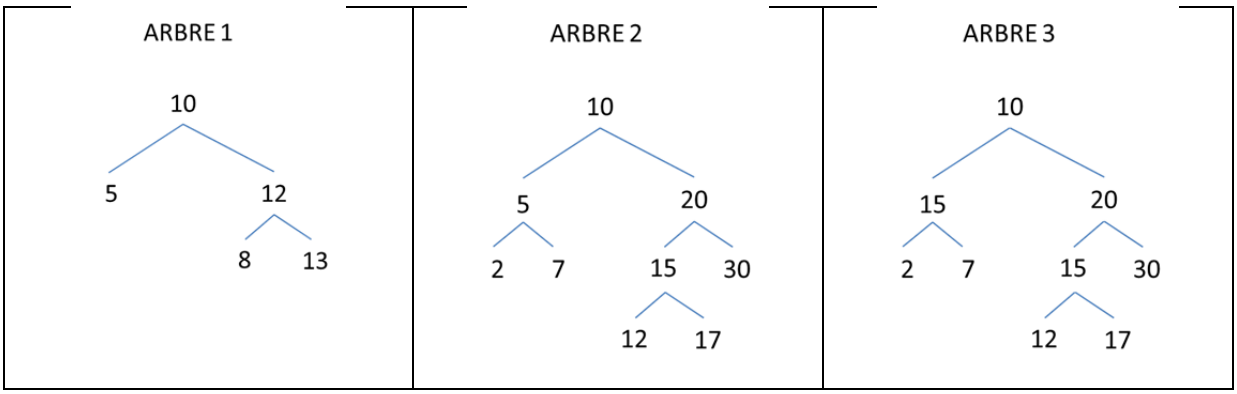
- Les éléments contenus dans tous les nœuds de son sous-arbre gauche ont des valeurs inférieures ou égales à celle de l'élément contenu dans le nœud courant.
- Les éléments contenus dans tous les nœuds de son sous-arbre droit ont des valeurs supérieures à celle de l'élément contenu dans le nœud courant.

Définition récursive d'un ABR :

Arbre Binaire de Recherche = ø
Si Arbre Binaire de Recherche = <r, arbre_g, arbre_d>, alors :

- pour tout x nœud dans l'arbre arbre_g, on a : contenu(x)≤contenu(r)
- pour tout x nœud dans l'arbre arbre_d, on a : contenu(r)<contenu(x)
  et
- arbre_g et arbre_d sont des ABR.

Exemple :



L'arbre binaire A2 <10, <5, <2>, <7>>, <20, <15, <12>, <17>>, <30>> > est un arbre binaire de recherche.

L'arbre binaire A3 <10, <15, <2>, <7>>, <20, <15, <12>, <17>>, <30>> > n'est pas un arbre binaire de recherche, parce que non(15≤10), non(15<7), même si (7≤10).

L'arbre binaire A1 <10, <5>, <12, <8>, <13>> n'est pas un ABR parce que non(10<8) alors que 10 est à la racine

Si A est un ABR, alors gauche(A) et droit(A) sont des ABR… mais l'exemple de Arbre1 nous prouve qu'il ne s'agit pas d'une condition suffisante…
Il faut en réalité que le plus grand élément du sous-arbre gauche soit plus petit que la racine et que le plus petit élément du sous-arbre droit soit plus grand que la racine.

Définition d'un ABR :

A est un Arbre Binaire de Recherche si et seulement si :
[A est vide] ou
[A est une feuille] ou
| A = <r, g(A), d(A)>
| **et** MAX(g(A)) ≤ contenu(r)<MIN(d(A))
| **et** g(A) et d(A) sont des ABR.

On peut assez facilement vérifier qu'un arbre est un ABR en utilisant un parcours infixe : si l'on écrit les valeurs de tous les nœuds d'un ABR en utilisant une procédure infixe, on obtient une liste ordonnée.

Si un arbre A est un ABR, alors gauche(A) et droit(A) sont des ABR ET Le plus grand élément du sous-arbre gauche est plus petit que racine(A) et le plus petit élément du sous-arbre droit est plus grand que racine(A)

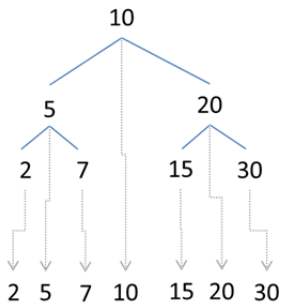
2. Écrire ou lister les valeurs des contenus des nœuds

Lors du parcours infixe d'un ABR, pour tout nœud courant, tous les nœuds de l'arbre gauche sont écrits avant le nœud courant, qui leur est supérieur ou égal. Tous les nœuds de l'arbre droit sont écrits après le nœud courant, qui leur est inférieur :

Infixe(A) = Infixe(g(A)) – [valeur de la racine (A)] – Infixe(d(A))

Par récurrence sur la taille de l'arbre, on peut montrer que, si A est un ABR, la liste rendue par Infixe(A) est ordonnée en ordre croissant.
La réciproque est vraie si tous les éléments de A portent des valeurs distinctes.

Pour bien visualiser cela, il suffit de penser qu'un parcours infixe correspond à projeter l'arbre sur une ligne imaginaire horizontale, puis à lire de gauche à droite.



Pour savoir si un arbre binaire (dont les éléments sont distincts) est un ABR, il suffit de vérifier que la liste obtenue par un parcours infixe est ordonnée.

Rappel du parcours infixe d'un arbre, ici en représentation chaînée.

```
Procédure Parcourir_Exposer_Infixe (A : Arbre )
    Si A ≠ Nil alors
        Parcourir_Exposer_Infixe (A.g)
        Écrire (A.val)
        Parcourir_Exposer_Infixe (A.d)
Fin Procédure
```

Le résultat attendu sera ici la liste ordonnée des nœuds, sans modification de l'arbre initial : de nouvelles places seront réservées pour constituer cette liste (on dit aussi « hors place »).

La construction de la liste infixe d'un ABR en hors place mime directement la formule de définition de la liste infixe : Infixe(A) = Infixe(g(A)) – [valeur de la racine (A)] – Infixe(d(A))

```
Fonction liste_infixe (A : Arbre) : Liste
                                {fournit la liste des nœuds dans l'ordre infixe}
                        {si A est un A.B.R. alors la liste écrite sera ordonnée}
Si A = Nil alors rendre Nil
Sinon rendre concat ( liste_infixe(A.g), cons(A.val, liste_infixe(A.d)) )

Fin Fonction
```

Si l'on considère que la complexité de cons est k (constante), la complexité de liste_infixe est donnée en fonction de la taille de l'arbre (nombre de nœuds) passé en argument par :

$$\text{complexité}(A) = \text{complexité}(A.g)+k+\text{complexité}(A.d)$$
$$= k*\text{taille}(A)$$

On peut démontrer cela par récurrence :

Complexité(0) = 0
Complexité(1) = k
Complexité(m) = k + Complexité(m-1)
d'où Complexité(m) = k*m, avec m la taille de l'arbre (*i.e.* en la taille des données).

3. Recherche d'un élément dans un ABR

Lorsqu'on recherche un élément *e* dans un ABR, la première étape, si l'arbre est non vide, est de comparer cet élément à la valeur portée par la racine. Cette première comparaison aiguille ensuite la recherche : si *e* est plus petit que l'élément porté par la racine, on continuera la recherche seulement dans le sous-arbre gauche ; s'il est plus grand, on le recherchera seulement dans le sous-arbre droit.

Voici un algorithme possible, qui renvoie un booléen.

```
Fonction Recherche(x:élément ; A: Arbre) : Booléen
    Si A=Vide alors rendre FAUX
    Sinon   Si x = contenu(racine(A)) alors rendre VRAI
            Sinon   Si x<contenu(racine(A)) alors
                            rendre Recherche (x, gauche(A)
                    Sinon rendre Recherche(x, droite(A))
```

Exercices pour s'entraîner :

- modifier l'algorithme ci-dessus pour qu'il retourne l'adresse de l'élément s'il est dans l'arbre, ou Nil s'il ne s'y trouve pas.
- transformer la fonction ci-dessus en procédure (donc dérécursifier).
-

Complexité de l'algorithme de recherche d'un élément dans un ABR

Si la recherche s'est terminée positivement en un nœud n, le nombre de comparaisons est égal à :

2*prof(n)+1

Si elle s'est soldée par un échec, ce nombre est 2*prof(n)+1, le nœud n étant le successeur de celui où s'est terminée la recherche.

Si tous les éléments sont distincts, la complexité est linéaire en la hauteur de l'arbre ; elle est donc au moins logarithmique en la taille de l'arbre, linéaire dans le pire cas (cas dégénéré, quand l'arbre est filiforme et l'élément à chercher au bout) et constante (meilleur des cas) si l'élément est à la racine.

C'est pour cette raison que l'on cherche en général à produire des arbres assez équilibrés ou à les ré-équilibrer.

Pourquoi une complexité logarithmique ?

Si un ABR, A_{compl}, est tel que tous ses nœuds non feuilles ont exactement deux fils et que tous les niveaux sont pleins, il comporte alors :

- au niveau 0 1 nœud 2^0
- au niveau 1 2 nœuds 2^1
- au niveau 2 4 nœuds 2^2
- …
- au niveau h 2^h nœuds

Le nombre de feuilles est donc de 2^h et sa taille (le nombre total de nœuds, qu'il s'agisse ou non de feuilles) est : $Taille(A_{compl}) = 2^0 + 2^1 + 2^2 + 2^3 + 2^h = 2^{1+h}$

La hauteur de l'arbre est donc :

$Hauteur(A_{compl}) = \log(taille+1) - 1$

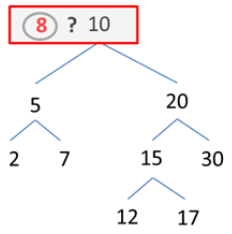
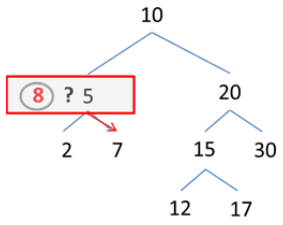
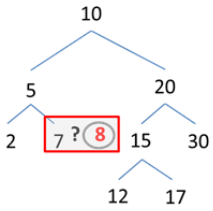
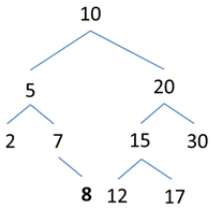
C'est la plus petite hauteur possible pour un ABR exprimée en fonction de sa taille (nombre de nœuds)

4. Ajouter un élément et une place en une feuille dans un ABR

On utilise beaucoup les ABR (sous leur forme équilibrée), notamment dans les applications de traitement de grandes masses données : dans ce contexte, nombre d'algorithmes construisent à la volée des arbres *ad hoc*.

Nous allons examiner la construction d'un ABR par adjonction de feuille.

Imaginons que l'on veuille ajouter l'élément de valeur 8 dans l'ABR Arbre2 de l'exemple précédant. On suivra alors les étapes suivantes.

			
Comparer 8 à la valeur de la racine. Il faut descendre à gauche.	Comparer 8 à 5 (2ème nœud). Il faut descendre à droite.	Comparer 8 à 7 (3ème nœud). Il faut descendre à droite.	On est positionné sur un arbre vide, on y positionne 8, comme racine d'un nouvel arbre sans fils.

Le principe de base est donc de descendre récursivement dans l'arbre en laissant à droite ce qui est plus grand que l'élément à insérer et à gauche ce qui est plus petit. On insère lorsqu'on atteint un arbre vide.

Exemple d'algorithme récursif d'ajout d'un élément en feuille.

```
Procédure Ajouter(x : élément ; var A : ABR)
                        {ajoute à un ABR une nouvelle feuille contenant x}
                                                {résultat : un ABR}
Si A ≠ vide alors
        si x ≤ contenu(racine(A)) alors
                Ajouter(x,g(A))
        sinon Ajouter(x,d(A))
Sinon accrocher à cette place l'arbre(x,vide,vide)
```

La complexité est linéaire en la hauteur de l'arbre, donc logarithmique en général en la taille de l'arbre (nombre total de nœuds), et linéaire en la taille de l'arbre dans le pire cas (cas dégénéré, quand l'arbre est filiforme et que l'élément à ajouter doit l'être au bout).

Exercice : écrire une version itérative de l'algorithme Ajouter ci-dessus.

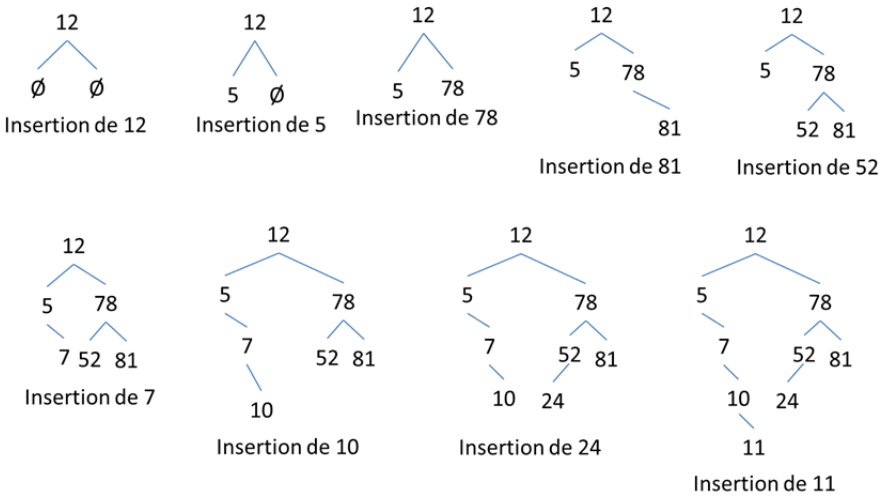
5. Construire un ABR par adjonction de feuilles

On peut évidemment utiliser la procédure ci-dessus pour construire des ABR par adjonctions successives de feuilles. Les valeurs sont lues (au clavier, dans un fichier, dans une liste fournie en entrée) et sont progressivement ajoutées à un arbre initialement vide.

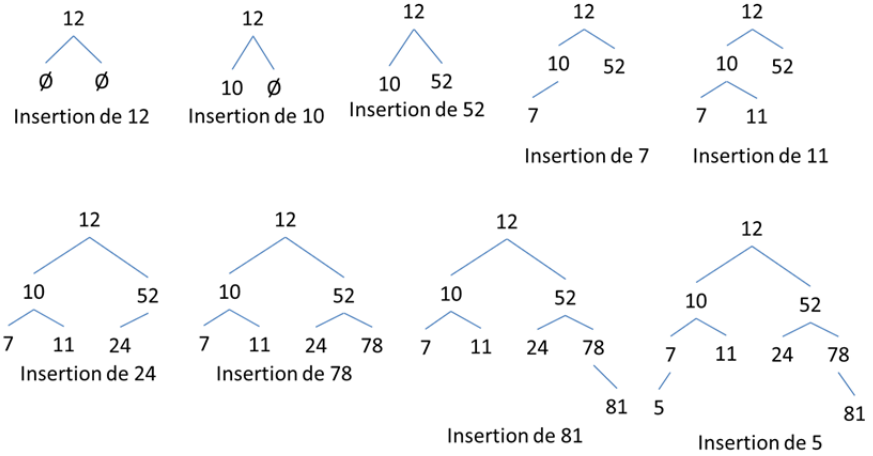
```
Procédure Construire(var A : ABR)
                        {construit un ABR en prenant des entrées clavier}
                                {par adjonctions successives de feuilles}
A <- NULL       {création de l'arbre vide}
Lire_Valeur(k)                                  {entrée clavier ou fichier}

Répéter
        Ajouter (k, A)
        Lire_Valeur(k)
Jusqu'à k = Défaut
```

Exemples de déroulement pour cette procédure

- Exemple 1 : lecture dans l'ordre de 12, 5, 78, 81, 52, 7, 10, 24, 11



- Exemple 2 : lecture dans l'ordre de 12, 10, 52, 7, 11, 24, 78, 81, 5



Les exemples 1 et 2 proposent l'insertion des mêmes valeurs. La seule différence est l'ordre dans lequel ces dernières sont lues et donc insérées…
On observe que l'arbre du second exemple est « complet » : tous les nœuds ont deux fils, sauf ceux portant les feuilles ; tous ses niveaux sont pleins, sauf celui des feuilles. En ce qui concerne l'arbre de l'exemple 1, aucun des niveaux n'est complet et le sous-arbre gauche est dégénéré.

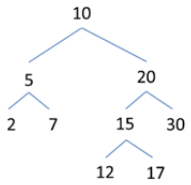
Nota Bene : Le résultat obtenu dépend des valeurs lues, mais également de l'ordre dans lequel elles l'ont été.

6. Supprimer un élément et une place d'un ABR

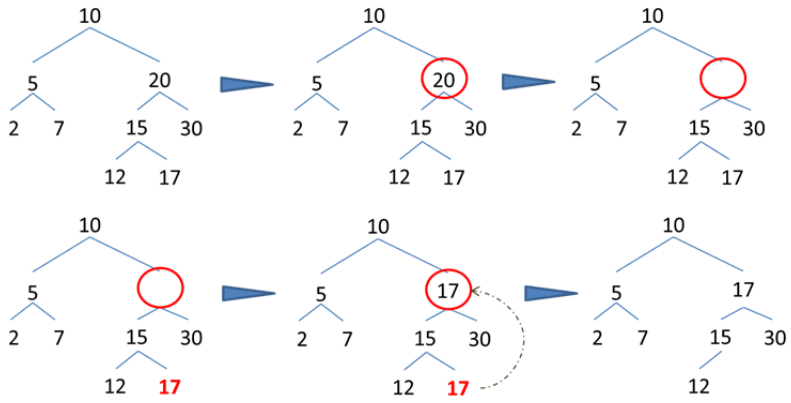
Il s'agit ici de retirer un élément de l'ABR tout en conservant la structure d'ABR et en n'ayant donc aucune place interne vide (seules les feuilles peuvent ne pas porter de valeur). Il faut donc :

1. Trouver l'élément à supprimer (déterminer sa place).
2. Supprimer la première occurrence de l'élément recherché.
3. Remplacer l'élément supprimé par un autre, judicieusement choisi.
4. Supprimer l'emplacement du remplaçant.

Exemple pour l'arbre suivant :



On souhaite supprimer l'élément de valeur 20.



On trouve la place contenant 20.
On remplace la valeur de ce nœud (20) par 17. **Cette valeur est choisie parce qu'elle est celle qui est immédiatement inférieure à 20 dans l'arbre.**
En clair : 17 est le plus grand élément du sous-arbre gauche en dessous de 20.
L'élément 17 se trouve ici sur une feuille, facile à supprimer sans dommage pour le reste de la structure.
Si ce n'avait pas été une feuille, il aurait fallu remplacer le nœud par son fils gauche.

Algorithme de suppression d'un élément et de sa place.

On veut créer une procédure Sans-Max qui rend l'arbre privé de la place et de la valeur de Max.
On sait que la plus grande feuille d'un ABR est la dernière feuille la plus à droite.

Algorithme de la procédure Sans-Max

```
{Cette procédure enlève de l'arbre A supposé non vide sa plus grande feuille,
        la place associée et récupère la valeur du maximum dans la variable v}

Procédure Sans-Max (Var A : arbre ; var v : élément)
    Si A.d est vide alors                    {Plus rien à droite : on est au bout}
        v ← A.val                            {on récupère la valeur}
        A ← A.g          {On enlève la racine et on garde ce qui est à gauche}
    Sinon Sans-Max (A.d, v)                  {Sinon on va chercher à droite}
```

On peut voir ici que la variable v récupère la valeur portée par l'emplacement le plus à droite avant qu'on supprime l'emplacement.

A partir de cette procédure, il est maintenant possible de supprimer l'élément x dans un ABR tout en lui conservant sa qualité d'ABR.

Algorithme de suppression d'un élément dans un ABR en conservant ses propriétés

```
Procédure Supprimer (x : élément ; var A : Arbre)
                                        {On suppose que A est un A.B.R.}
Si A n'est pas vide alors               {on cherche x}
    Si x < val(A) alors
        Supprimer (x, A.g)              {on le cherche à gauche}
    Sinon
        Si x > val(A) alors
            Supprimer (x, A.d)          {on le cherche à droite}
        Sinon                           {on l'a trouvé}
            Si A.g est vide alors
                A←A.d                   {on remonte l'arbre droit}
            Sinon
                Si A.d est vide alors
                    A←A.g               {ou on remonte le gauche}
                Sinon Sans-Max(A.g, v)          {on ampute}
                    A.val←v
                      {la dernière feuille est mise à la place de x et
                        on enlève la place de cette dernière feuille}
```

La complexité est assimilable à celle du parcours, donc linéaire en la hauteur de l'arbre, c'est-à-dire logarithmique en général en la taille de l'arbre (*i.e.* le nombre de nœuds internes et externes)...
Dans le pire des cas, la complexité est linéaire en la taille des données, dans le cas d'arbre dégénéré (filiforme) dans lequel l'élément à supprimer est plus petit que l'extrémité gauche ou plus grand que l'extrémité droite.

7. Vérifier qu'un arbre est un ABR

Il s'agit ici d'un exercice d'école : écrire une fonction testant si l'arbre passé en paramètre est un ABR.
On parcourt l'arbre suivant le procédé infixe : les nœuds rencontrés doivent être ordonnés, comme indiqué plus haut.

Les étapes sont les suivantes :

1. Vérifier que le sous-arbre gauche est un ABR
2. Retenir le plus grand élément du sous-arbre gauche
3. Comparer le plus grand élément du sous-arbre gauche et la racine
4. Vérifier que le sous-arbre droit est un ABR
5. Retenir le plus petit élément du sous-arbre droit
6. Comparer le plus petit élément du sous-arbre droit avec la racine

Pour enchaîner toutes ces étapes, on recourra à une fonction appelant une procédure récursive.

Fonction englobante :

```
                                {Cette fonction vérifie que A est un ABR, dans le cas
                                    où tous les éléments sont distincts}
Fonction Est_ABR(A : Arbre) : booléen
                                /*variables locales avant : entier ; stop : booléen*/
    avant←minimum_absolu
        {« avant » sera à chaque étape la plus grande des valeurs rencontrées}
    stop ← faux
      {« stop » sera mis à vrai quand il y aura eu violation de la propriété A.B.R.}
    Es_tu_un_ABR(A, avant, stop)
    rendre non stop
Fin Fonction
```

Procédure récursive :

```
Procédure Es_tu_un_ABR (A : Arbre ; var avant : entier ; var stop : booléen)
    Si non (vide ?(A)) et (non stop) alors
        Es_tu_un_ABR(g(A), avant, stop)        {on descend à gauche}
        Si (non stop) alors
            si avant > val(racine (A)) alors
                stop←vrai                      {si échec, on s'arrête}
            sinon                              {si succès, alors aller voir à droite}
                avant←val(racine (A))
                Es_tu_un_ABR(d(A), avant, stop)
            Si avant < val(racine (A)) alors
                stop←vrai                      {si échec, on s'arrête}

Fin Procédure
```

Les paramètres avant et stop sont passés par adresse (ou référence) et leurs modifications sont donc transmises au retour de la procédure appelante, ce qui garantit l'arrêt du processus.